# Evolution of an SGML Application Generator

Lynne A. Price
Joe Schneider
*Hewlett-Packard Co.*

**Abstract. The Standard Generalized Markup Language (SGML) is a notation for describing classes of structured documents and for coding documents belonging to described classes. An advantage of SGML and other grammar-based document representations is the ability to perform multiple applications on a single document source file. This paper describes the evolution of a software development tool for implementing such applications. It explains the original design as well as enhancements made during the system's first eighteen months. Although not statistically significant, data on the use of the enhanced features are presented. The experience described is relevant to other software engineering tools for text processing.**

The Standard Generalized Markup Language (SGML) is described in International Standard 8879 [1], published in late 1986. SGML evolved from macro-based word-processing and text-formatting tools. It encourages a writer to use **descriptive markup**, identifying structures within a document, rather than **procedural markup**, specifying processing. For example, "this is a section heading" is preferred to "center this line in boldface." The philosophy underlying descriptive markup has been repeatedly described [2, 3]. Its use allows a single source document to be processed by various applications. The text can, for example, be formatted, loaded into an online database, or analyzed for various linguistic properties.

This paper discusses the evolution of a set of software development tools for supporting SGML applications. This material is introduced by an overview of SGML and a description of the original design of the software support system.

## Principles of SGML

SGML recognizes that no finite set of structural elements can account for the vast flexibility permitted in written texts. Instead, it provides features for defining types of documents and then coding particular documents that belong to the defined types. SGML, thus, is one grammar-driven approach to structuring documentation [4]. Possible document types include reference manuals, user's guides, product specifications, memos, and letters. A document type is formally defined with an SGML **document-type definition** which itemizes the structural elements that can occur in documents of that type and defines the contexts in which each element can occur. Most document-type definitions are prepared by a small group of individuals for many more people to use with a large number of documents. Thus, most SGML users create and edit documents rather than document-type definitions.

The bulk of SGML markup is used to mark the beginning or end of structural elements. Elements are delimited with **tags** containing the element name enclosed by single-or multi-character delimiter strings. Different delimiters are used for the start-tags that open elements and the end-tags that close them. **Start-tags** in this paper are enclosed by angle brackets while **end-tags** are enclosed by </ and >.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and / or specific permission.

© 1988 ACM 0-89791-291-8/88/0012/0051 $1.50

The document-type definition also controls various abbreviatory conventions, or **markup minimization**. One form of minimization lets a writer omit a start-tag or end-tag when the boundary of the delimited element is clear from context. If it is known, for example, that every chapter begins with a chapter title, the SGML processor can recognize the first words in a new chapter as the title whether or not the writer has included the title start-tag. Similarly, since the end of a chapter implies the end of its last paragraph, the paragraph end-tag is optional. Context-sensitive interpretation of particular character strings is also possible. For instance, an asterisk may be interpreted as a code for the multiplication symbol inside a mathematical formula but as a footnote indicator elsewhere.

Consider a document-type definition that specifies rules for creating a glossary consisting of a one-line title and multiple entries. Each entry consists of a one-line term followed by a definition. Definitions, terminated with blank lines, may contain cross-references to other terms in the glossary. The beginning of such a glossary might be coded as in Figure 1.

```
Glossary of Animals

Aardvark
The first animal listed in a dictionary.

Dog
A domesticated canine, probably descended
from the <xref>wolf</xref>.
.
.
.
```

**Figure 1. An SGML parser uses the rules in the accompanying document-type definition to recognize the title, defined terms, and definitions in this sample SGML source file. The author begins each entry by placing the defined term on its own line and separates successive entries with one or more blank lines. There is no need to delimit the structural elements more explicitly.**

Note that the only explicitly delimited element is the cross-reference to the entry `wolf` in the definition of `dog`. The conventions for markup minimization determine the remaining structure. The original input file is equivalent to the more cumbersome form shown in Figure 2.

Existence of the document-type definition allows an SGML parser to report errors in document structure. Stylistic conventions, such as requiring at least one introductory paragraph before the first figure in a chapter, can be enforced. Errors familiar to users of batch formatting systems are automatically localized. For instance, the SGML processor will inform the writer when an emphasized phrase spans multiple paragraphs or an itemized list extends past the end of a chapter.

```
<glossary>
<title>Glossary of Animals</title>

<entry>
<term>Aardvark</term>
<definition>The first animal listed in a
dictionary.</definition>
</entry>

<entry>
<term>Dog</term>
<definition>
A domesticated canine, probably descended
from the <xref>wolf</xref>.
</definition>
</entry>
.
.
.
</glossary>
```

**Figure 2. The SGML parser treats the source file of Figure 1 exactly as though the beginning and end of each element were tagged. In general, there is no need to generate this completely expanded markup, even automatically.**

## MARKUP and the HP Tag Project

Over fifty separate writing departments located throughout the world produce user guides and reference manuals for Hewlett-Packard computers, software, and electronic instruments. Electronic interchange of material between departments is often desirable; for example, a manual written in one country may be translated to the local language in another. In the past, interchange has been complicated by an assortment of text processing tools and the corresponding differences in markup as well as by the diverse hardware on which the tools were installed.

SGML supplies a means of standardizing markup conventions throughout the company. A shared markup technique also provides a vehicle for discussion among writers in different groups. In mid-1986, therefore, Hewlett-Packard began developing a general purpose SGML parser and application generator, called MARKUP, for in-house use [5]. Since the environment already provided powerful formatting and composition tools, the software development effort was reduced by using SGML as a front end to existing text processing systems. In this way, files can be interchanged without replacing existing software and the corresponding hardware.

HP Tag, an SGML document-type definition for Hewlett-Packard user documentation, was written in parallel with the software development project. MARKUP simplified prototyping and revision of the HP Tag definition. It also supported development of multiple applications for HP Tag documents.

The first HP Tag application, used for formatting and typesetting, was released to the company's technical writers, editors, and illustrators on two different computer systems early in 1988. In addition, HP Tag documents can be input to a computer-assisted translation system. MARKUP and HP Tag are also used to prepare data for LaserROM, a commercially available CD ROM package for user manuals and support information [6].

The MARKUP approach has proven its utility during development of these first HP Tag applications. The remainder of this paper describes enhancements that have been made as the software development environment matures.

## System Overview

MARKUP consists of an SGML parser and facilities for defining applications on SGML documents. MARKUP is appropriate for the construction of "down translators" that convert standard-form documents to a nonstandard form [7]. It can be used to develop SGML front ends to available tools as well as stand-alone applications.

Figure 3 models the use of MARKUP as a front end to an existing formatter. In the first of two stages of processing, MARKUP analyzes the markup. It verifies that all elements occur in appropriate contexts and that the document matches the document-type definition. It also expands all markup minimization. It outputs an intermediate form of the document, the text marked-up according to the notation of the target formatter. In the second stage, this intermediate form is processed by the target formatter to actually prepare the hard copy.

To prepare text for formatting, the writer identifies its structural elements without creating procedures for typesetting those elements. The MARKUP processor automatically generates the typesetting directions required by a particular target formatter to process the structures identified in the source file. To make this translation, MARKUP relies on information provided by two special users, the **document-type designer** and the **interface designer**. Their roles are as follows:

1. The document-type designer specifies what structures can occur in what contexts.
2. The interface designer specifies how to prepare each structure for a particular text formatter and output to a particular device.

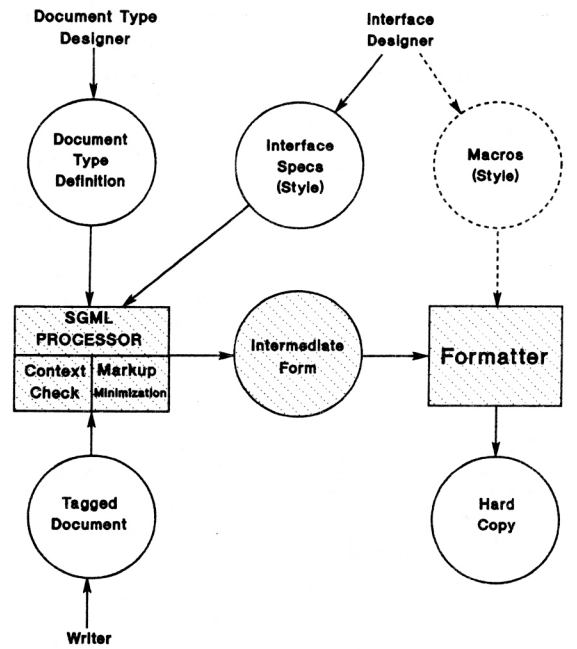


**Figure 3. The writer submits an SGML-tagged document to a MARKUP application and obtains a formatted hard-copy version prepared by an external document preparation system. The writer need not distinguish the intermediate steps indicated by the shaded blocks. Before the document is formatted, a document-type designer specifies the document-type definition and an interface designer specifies the translation from SGML. Dotted lines indicate optional use of the formatter's macro facility.**

The document-type designer and interface designer may, of course, be the same person. Furthermore, once a document-type definition exists, interfaces between it and different target formatters can be written by different people, each knowledgeable about a different formatter.

Translation from the SGML form of the document to that required by the target formatter is dependent on information supplied by the document-type designer and interface designer. Thus, input from these sources is required before a writer can use the result. If the target formatter supports a macro facility, the interface designer may choose to use it. The style—fonts, spacing, etc.—of the output document may be encoded directly in the interface or in the macro package, if any.

The MARKUP system contains three programs:

1. BUILD processes document-type definitions.
2. ELTDEF processes interface specifications for a particular document type. (The

program name is a contraction of "element definition.")

3. PARSER processes documents coded for a particular document type and produces the output defined in a particular interface specification. PARSER, thus, performs an SGML application as well as parsing SGML documents.

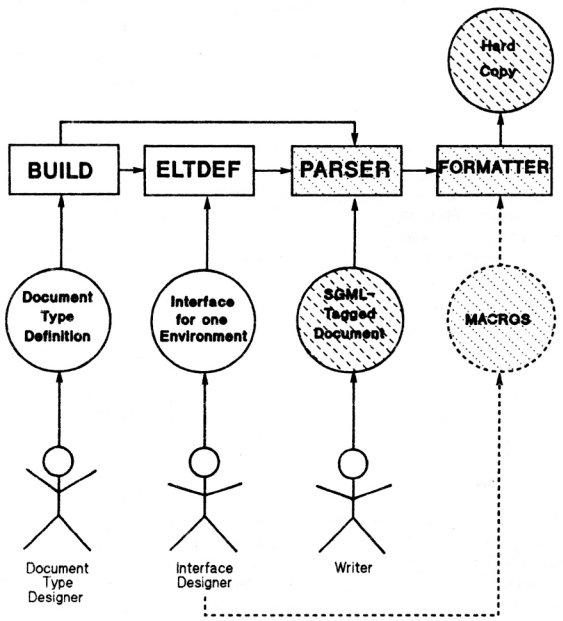


**Figure 4. Data flow through the three programs of the MARKUP system is diagrammed above. Only shaded elements need be installed on the writer's computer system. The writer need not distinguish among components indicated in the lighter shaded. Dotted lines again indicate optional elements.**

The use of the three programs with a target formatter is shown in Figure 4. As indicated, the results of BUILD are fed into both ELTDEF and PARSER. The results of ELTDEF are used by PARSER. Only the items shaded in the figure—PARSER, the target formatter, any macros used, and, of course, the user's data—need be installed on the particular computer system used by the writer. From the writer's view, PARSER, the formatter, and any macro package form a single package. A more detailed view of the same relationships is given in Figure 5, where the stick figures represent users of the three programs.

In fact, the output of BUILD is a customized version of the ELTDEF program that processes interface definitions only for one particular document type. The output of ELTDEF is a version of PARSER specific to the processed document type and interface definitions. This scheme is indicated in Figure 6. As shown, different versions of PARSER for the same document type can process the same documents. The output, however, varies with the interface.

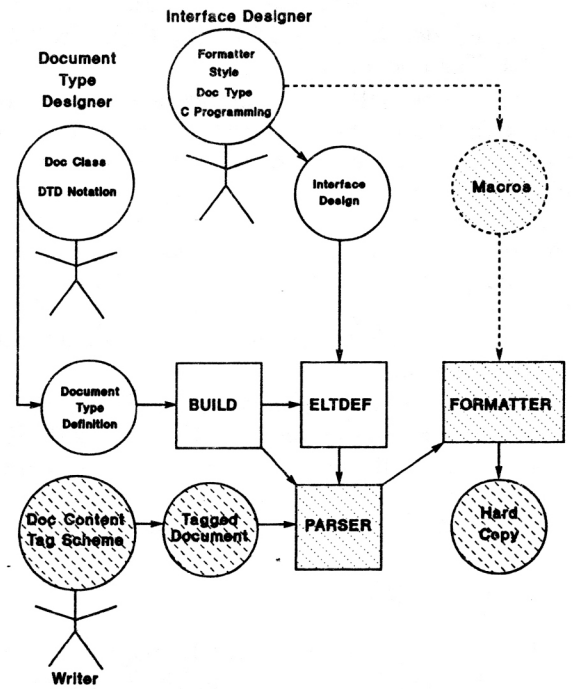


**Figure 5. An indication of the knowledge required to use each MARKUP program is shown within the circle representing the corresponding user's head. The writer must be familiar with the tagging scheme being used—the identifiers used to label structural elements—and with the topic he is describing. The document type designer must understand the class of documents he is describing and the MARKUP notation for defining document types. The interface designer must know the desired style of output, how to use the target formatter, and the document type. In most cases, he must have some programming skill as well.**

The interface definition makes MARKUP a powerful tool for efficient implementation of certain types of SGML applications. It is inappropriate, however, in the following cases:

- MARKUP does not support interactive applications because it relies on sequential processing of the input stream. It is therefore not an appropriate tool for SGML-based text editing.
- MARKUP requires a different interface specification for each document-type definition and hence would be tedious to use while developing applications relevant to all possible document-type definitions.
- SGML allows each document to specify its document-type definition, either in entirety or as a modification of an existing one. MARKUP assumes that the document-type

definition is externally defined so that the application can be compiled before a document is processed. Thus, another development approach is needed in environments where the document-type definition cannot be specified in advance.

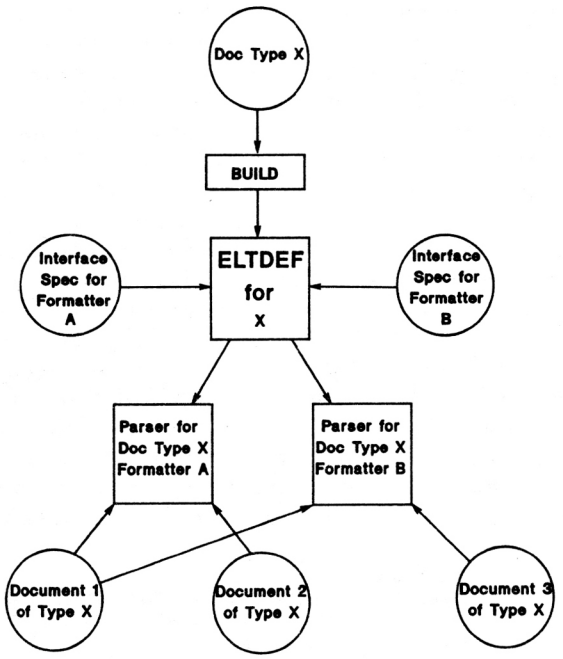
Doc Type X
BUILD
Interface Spec for Formatter A
ELTDEF for X
Interface Spec for Formatter B
Parser for Doc Type X Formatter A
Parser for Doc Type X Formatter B
Document 1 of Type X
Document 2 of Type X
Document 3 of Type X

**Figure 6. Circles indicate data files and rectangles indicate programs in this model of an environment where documents of a particular document-type definition (X) are submitted to two SGML applications (called A and B).**

## The Original Design of MARKUP

The essence of SGML is the division of a document into structural elements. The essence of a MARKUP application definition is the description of actions to be performed when specified elements occur. These actions are triggered whether the element is delimited by explicit tags or is detected through the markup minimization algorithm. Thus, in contrast to some other approaches to defining SGML applications, MARKUP does not require generation of an unminimized file before other processing is performed.

In a technique modeled after parser generators such as *yacc* [8], many actions are segments of C source code that are automatically executed at specified points in the document. These code segments may be single function calls or any number of C statements. MARKUP provides a mechanism for declaring global variables as well as subordinate procedures that are accessible to all code segments.

The bulk of an interface definition is a sequence of **element blocks**. Each element block specifies actions to be performed at the beginning and end of each occurrence of the specified element, as well as processing of data characters occurring within them. All fields within the element block are optional and element blocks may be omitted for elements if no action need be taken when they occur.

```
<ELEMENT GLOSSARY>
   <START-STRING>\input glossmac
   </START-STRING>
     <END-STRING>\bye
   </END-STRING>

   <ELEMENT TITLE>
     <START-STRING>\title{</START-STRING>
     <END-STRING>}
   </END-STRING>

   <ELEMENT TERM>
     <START-STRING>\defterm{</START-STRING>
     <END-STRING>}
   </END-STRING>

   <ELEMENT DEFINITION>
     <END-STRING>

   </END-STRING>

   <ELEMENT XREF>
     <START-STRING>{/it </START-STRING>
     <END-STRING>}</END-STRING>
```

**Figure 7. This input to ELTDEF defines a TEX macro scheme for typesetting the glossary example of Figures 1 and 2.**

Code segments triggered at the start and end of elements are called **start-code** and **end-code**. Frequently, the desired action is outputting a formatter instruction. To spare the interface designer the awkwardness of the C syntax required to output such strings in code segments, MARKUP provides a **start-string** and **end-string** to address this special case. Although the interface definition in general is free-format, a carriage return within one of these strings is significant and specifies insertion of a carriage return in the output file.

By default, MARKUP copies data characters that occur in a document from the input stream to the output file. An element block can contain **text-code**, which specifies the processing of data characters that occur within the element and its subelements. Examples of the use of text-code include automatically capitalizing words in a title, condensing a sequences of blanks to a single space,

and entering specified items into an index or table of contents.

As an example, consider an application that prepares glossaries of the form discussed earlier for typesetting by TEX [9, 10]. A simple interface to perform this translation is shown in Figure 7. In the sample definition, no TEX instruction is needed for the `entry` element and this element has no element block. The resemblance of the interface specification to SGML markup is cosmetic only. Figure 8 gives corresponding output for the sample glossary. Identical output is produced from the minimized input of Figure 1 and the expanded markup of Figure 2.

```
\input glossmac
\title{Glossary of Animals}
\defterm{Aardvark}
The first animal listed in a dictionary.

\defterm{Dog}
A domesticated canine, probably descended
from the {\it wolf}.
                .
                .
                .
\bye
```

**Figure 8. The version of PARSER defined in Figure 7 produces a source file that can then be submitted to TEX.**

## Evolution of MARKUP

As the first MARKUP applications developed and grew, interface designers began to request enhancements. These enhancements in many cases simplify interface specification without extending the system's capabilities. For instance, it was found that, before outputting character strings, interfaces frequently test either the context in which a particular construct occurs or the state of various flags. The initial design provided no mechanism for performing these tests within start-strings and end-strings, so the more cumbersome start-and end-code were used.

The new MARKUP design provides methods for performing tests while using the string constructs. Similarly, it is now possible for a single action to be invoked in multiple contexts. Other changes simplify error recovery.

As a result, an interface designer finds MARKUP more convenient and is more likely to implement certain types of processing. Other text processing development systems (whether or not based on SGML) should consider providing analogous features so that design decisions can be based on the functionality provided to the user rather than the effort required by the programmer.

Major changes made to MARKUP in the year and a half since the system was first released are itemized below:

**Context-Sensitive Actions.** Many document-type definitions declare elements that can occur in different contexts. Chapters, sections, and appendices, for instance, may all require headings. Processing of such elements may depend on the environment in which they occur. Interfaces written for the initial version of MARKUP frequently contained repeated tests for context. Since such testing is impossible within start-strings and end-strings, start-code and end-code were required when different strings were needed in different contexts. The enhanced MARKUP allows a list of element names to be included with an action to indicate that the action should be applied only within the specified elements. For example, the following indicates that a `head` within a `chapter` is output surrounded by `\newchap{` and `}` but that a `head` within a `section` is output surrounded by `\newsect{` and `}`:

```
<ELEMENT HEAD>
  <START-STRING CHAPTER>\newchap{</START-STRING>
  <START-STRING SECTION>\newsect{</START-STRING>
  <END-STRING>}<END-STRING>
```

If the context specification contains several elements, they are listed with the innermost element on the left and containing elements on the right. Thus, the following specification calls function `f1` after a `head` in a `section` in a `chapter`, but `f2` after a `head` in a `section` in an `appendix`:

```
<ELEMENT HEAD>
  <END-CODE SECTION CHAPTER>f1() ;
  <END-CODE SECTION APPENDIX>f2() ;
```

An element name can appear several times within a single context specification. For example, if a document contains a set of three nested lists, the following code will call indent with the argument `1` at the start of each item in the outermost list. At the start of each item in the next level of nesting, an `item` in a `list` in a `list`, `indent(2)` will be executed. At the start of each item in the innermost list, the value `3` will be passed to `indent`:

```
<ELEMENT ITEM>:
  <START-CODE LIST>indent(1) ;
  <START-CODE LIST LIST>indent(2) ;
  <START-CODE LIST LIST LIST>indent(3) ;
```

When more than one context specification matches the context in which an element occurs—as all three specifications in the previous example match an item in the innermost of three nested

lists—the action with the greatest number of elements is used. If multiple actions have the greatest number of elements, the first to occur in the interface definition file is selected.

The addition of context-sensitive actions simplifies error recovery strategy for applications that attempt to process malformed documents. Although PARSER always reports structure errors to the user, actions specified in an interface are invoked even if the element occurs illegally. Suppose a document-type definition allows `rows` only inside `tables`. The two code segments

```
<ELEMENT ROW>
  <START-CODE TABLE> action() ;
```

and

```
<ELEMENT ROW>
  <START-CODE> action() ;
```

will produce the same results in a document with no errors. The first form suppresses the call to `action` in an erroneous document in which a `row` is entered out of context.

The context specification in an action pseudo-tag need not be complete. An asterisk indicates the possibility of intervening elements in the hierarchy. For example, consider a typesetting application in which chapters are formatted in a single column but appendices are done in double columns. Chapters and appendices may both be subdivided into sections and subsections. Footnotes can occur in both contexts, either in paragraphs or in tables. The following element block accounts for these possibilities:

```
<ELEMENT FOOTNOTE>
   <START-CODE P * CHAPTER>
     onecol_fnote() ;
   <START-CODE P * APPENDIX>
     twocol_fnote() ;
   <START-CODE * TABLE>
     table_fnote() ;
```

Multiple context specifications, separated by commas, can be entered in one action specification. For example, the following definition invokes action `a` at the start of a head occurring within a note, caution, or warning, but action `b` is triggered at the start of a head in a paragraph.

```
<ELEMENT HEAD>
   <START-CODE NOTE, CAUTION, WARNING>
     a() ;
   <START-CODE P>
     b() ;
```

**String-Code.** As mentioned earlier, start-strings and end-strings provide a convenient shorthand for specifying a frequent type of start-code and end-code. **String-code** is an enhancement to this feature that extends the situations in which this abbreviation can be used. The interface designer uses string-code in an element block to define in C the processing of start-strings and end-strings of the specified element and of elements that occur within it.

Applications using string-code include those that process only selected portions of an input document. The string-code can output start-strings and end-strings within selected elements and discard them in unselected contexts. This technique can be used, for example, to suppress information such as answers to exercises when processing the student version of a text book whose source file contains information needed only for the teacher's edition.

**Global Actions.** The extensions to MARKUP introduced in this and following sections do make it possible to build applications that could not be written with the initial design. One early extension was the ability for the interface designer to specify actions taken before starting to process a document and after completing it. When the input is a valid document, the same results are obtained by invoking these actions at the start of the **document element** (the element that contains the entire document). However, if a document contains a structural error, MARKUP attempts to continue processing it. Thus, if a document-type definition specifies one element that only occurs within another, the interface designer can assume that in a correct document the start-code triggered by the containing element has been executed before the start-code of the nested element. If the input is incorrect, however, the outer element may never have occurred. The results depend on the application, but global start-code allows the interface designer to insure that basic processing such as opening files has occurred before code in any element block is activated.

**Code Entities.** A writer includes an externally defined object such as a graphics files or special character into an SGML document by referring to a construct called an **entity**. In the original design of MARKUP, writers could define entities corresponding to abbreviations for objects such as frequently entered strings, instructions for the target text formatter, graphics files, and external data files containing parts of the document source (include files).

An early enhancement to MARKUP was the ability for an interface designer to define an entity with C code. A simple example illustrates this feature. Suppose a writer refers to an entity which inserts the current date into the document. If MARKUP is used to prepare SGML documents for a formatter that provides a date function,

the original entity capability is sufficient. If the formatter does not provide this feature, however, the interface designer uses a **code-entity** to retrieve the date from the operating system and send the resulting string to the formatter.

**Processing Instructions.** A **processing instruction** is an SGML construct that allows a writer to define special processing to be performed at a specified point in a document. Processing instructions can be entered through entities or typed directly into the document source file. They provide the writer with an escape mechanism for augmenting the normal processing of an application. This capability should be used cautiously because it deviates from the goals of descriptive markup. In particular, processing instructions must be changed when a source file is submitted to a different application. Pragmatically, however, writers must be able to address those atypical requirements of a single document that are not reflected in the document-type definition and to circumvent shortcomings in application software.

The original MARKUP assumed processing instructions were commands for a target text formatter and simply copied them directly to the output file. This action remains the default for the current system. However, in some situations, other treatment of processing instructions is needed. Consider, for example, a document in which a processing instruction is used to represent a special character or icon. Suppose the symbol occurs in a string that the interface saves in an internal buffer, writes to the output file more than once, or sorts into an alphabetized list. MARKUP now allows **PI-code** to be specified in an element block so that the interface designer can define handling of processing instructions in C. PI-code applies to processing instructions whether they are entered directly into the source file or referenced through entities.

**Character Input.** The SGML Standard provides a mechanism for defining the computer character set in which a document is coded. This ability was ignored by the first release of MARKUP. The enhanced system is not able to process the SGML character set specification. However, the interface designer can provide C statements that calculate successive input characters. This **character-code** can be used to select among alternate character sets. It also allows MARKUP applications to process data stored in a compressed format or retrieved from a database management system.

**Feature Use in Existing MARKUP Applications**

To obtain an indication of whether the enhanced features are useful, five MARKUP interface designers were asked to report on the frequency with which they use various features. The results are summarized in Table 1. These data are insufficient to predict characteristics of an interface before it is written, especially since not all enhancements were available to all programmers during the design and development period. Large differences in the needs of different applications and the styles of different programmers, however, are shown.

Interfaces C and D use the HP Tag document-type definition; A and B use different variations of HP Tag. The same individual wrote interfaces D and E (which are used in the LaserROM project).

| Interface | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| Elements defined | 158 | 169 | 155 | 155 | 27 | 56 |
| Start-strings | 10 | 8 | 76 | 59 | 2 | 24 |
| End-strings | 15 | 5 | 81 | 52 | 6 | 27 |
| Start-codes | 126 | 70 | 127 | 22 | 24 | 26 |
| Context-sensitive start-codes | 26 | 20 | 47 | 0 | 0 | 0 |
| Asterisks in start-codes | 4 | 11 | 27 | 0 | 0 | 0 |
| End-codes | 17 | 44 | 89 | 4 | 9 | 14 |
| Context-sensitive end-codes | 18 | 0 | 36 | 0 | 0 | 0 |
| Asterisks in end-codes | 2 | 0 | 23 | 0 | 0 | 0 |
| Text-codes | 17 | 7 | 38 | 20 | 9 | 2 |
| PI-codes | 0 | 0 | 9 | 0 | 0 | 0 |
| String-codes | 0 | 0 | 4 | 0 | 0 | 0 |
| Start-strings with start-code | 2 | 6 | 37 | 15 | 0 | 0 |
| End-strings with end-code | 2 | 1 | 30 | 1 | 0 | 0 |
| Longest context specification | 10 | 2 | 2 | 0 | 0 | 0 |
| Code entities | 0 | 167 | 0 | 0 | 0 | 0 |
| Global actions used | yes | yes | yes | no | no | yes |
| Character-code used | no | no | yes | no | no | no |
| Approximate lines of code | 3000 | 6420 | 4500 | 1700 | 650 | 750 |

**Table 1. MARKUP programmers report using various features with different frequency.**

While the four HP Tag-related interfaces all use document-type definitions with about 160 elements, the other two document-type definitions are much smaller. Some general trends are nevertheless apparent. No action is needed at the start of many elements, but start-code and start-string are more heavily used than end-code and end-string. The three interfaces that use context-sensitive actions use the asterisk placeholder in many, but not all, context-sensitive start-codes. Most interfaces use start-code more heavily than start-string (possibly because string-code is a recent enhancement).

Differences are also apparent. Interface A lists as many as ten elements in a single context specification while interfaces B and C do not use more than two. Only four of the six interfaces use global actions. Interface B is the only one to use code entities, a result of the back end

formatter it uses. Reflecting the amount of processing performed by different applications on the same document-type definition, interface C contains more than two and a half times as many lines of code as interface D. Interface C also has many more elements for which both start-string and start-code (or end-string and end-code) are specified.

Only interface C is listed as using character-code, PI-code, and string-code, but new versions of MARKUP are released to this programmer first. Other programmers reported that given more time with the enhanced system, they would probably recode the existing interfaces in order to make better use of new system capability.

## Future Research and Conclusions

MARKUP is continuing to evolve. One planned enhancement will add interface-specific data to PARSER's stack of open elements. The item count for each level of a nested list, for instance, would then automatically be allocated as each new list is opened.

An area for further work is sharing interface code among applications on related document-type definitions. An include file capability for interface definitions is a planned step in this direction. This enhancement will simplify maintenance of formatting applications on a set of document-type definitions with different high-level structure but common text elements. For example, memos, letters, and technical reports may share portions of the document-type and interface definitions for paragraphs, lists, and tables. More elaborately, an interactive tool for maintaining interfaces would assist the developer in locating portions of an interface that must be modified when the corresponding document-type definition is changed.

Although MARKUP does allow multiple contexts to be specified in a single action (e.g., `<START-CODE P, LIST>`), the interface designer must rely on C's macro capability and function calls if the same code is needed in different types of actions (e.g., in text-code and end-code) or if it is repeated in different element blocks. The interface designer would benefit from a straightforward syntax that reduced repetition of common code segments.

Actions to be taken for every start-tag or every end-tag (regardless of the type of element delimited) would allow MARKUP to be used for some applications that are independent of a particular document-type definition. In the current system, as simple a process as counting the number of elements that occur in a document requires code for every type of element.

Analysis of existing interfaces may suggest further enhancements. When both a start-string and start-code apply to a given context, the current system executes the start-code first and then processes the start-string. Similarly, end-code is executed before end-strings. Examination of existing interfaces might suggest that strings should be processed before code or that a reverse order is appropriate at the start of an element than at its end.

The enhancements described in this paper have simplified programming and increased the functionality of possible applications. Both MARKUP and, more importantly from the user's viewpoint, the applications built with it, have been well received in the internal environment where they have been released. It is hoped that the experience described here will suggest techniques to designers of other tools for processing structured documents, whether based on SGML or other conventions.

## References

[1] International Organization for Standardization, Information Processing—Text and office systems—Standard Generalized Markup Language (SGML), ISO 8879-1986(E).

[2] J. H. Coombs, A. H. Renear, and S. J. DeRose, "Markup Systems and the Future of Scholarly Text Processing", *Communications of the ACM,* V. 30, 1987, pp. 933–947.

[3] L. A. Price, "Three Topics as SGML Moves from Theory to Practice," *Proceedings, Corporate Electronic Publishing Systems™—The Federal Conference,* May, 1987, pp. 133–146. "Markup: Hewlett-Packard's SGML Implementation" is reprinted in *<Tag> The SGML Newsletter,* V. 1, Issue 2, pp. 14–15.

[4] R. Furuta, "Complexity in Structured Documents: User Interface Issues", *PROTEXT IV: Proceedings of the Fourth International Conference on Text Processing Systems,* October 1987, Boston. J. J. H. Miller, editor, Boole Press, Dublin, pp. 7–22.

[5] L. A. Price, "A Parser Generator for SGML", *PROTEXT IV: Proceedings of the Fourth International Conference on Text Processing Systems,* October 1987, Boston. J. J. H. Miller, editor, Boole Press, Dublin, pp. 118–123.

[6] M. Rafeld, "The LaserROM Project: A Case Study in Document Processing Systems", *Proceedings, ACM Conference on Document Processing Systems,* December 1988, Santa Fe.

[7] S. A. Mamrak, et. al., "A Software Architecture for Supporting the Exchange of Electronic Manuscripts", *Communications of the ACM*, V. 30, 1987, pp. 408–414.

[8] S. C. Johnson and M. E. Lesk, "Language Development Tools," *Bell System Technical Journal*, V. 57, 1978, pp. 2155–2175.

[9] D. E. Knuth, *The TEXbook*, Addison-Wesley, Reading, MA, 1984.

[10] L. A. Price, "SGML and TEX," *TUGBoat: The TEX Users Group Newsletter*, American Mathematical Society, V. 8, 1987, pp. 221–225.